

# Recorrido sobre árboles

## BFS (Breadth-First Search)

- Se comienza por el nivel 0 (la raíz) y se visita cada vértice en un nivel antes de pasar al siguiente.
- LISTA implementada como **cola**.

## DFS (Depth-First Search)

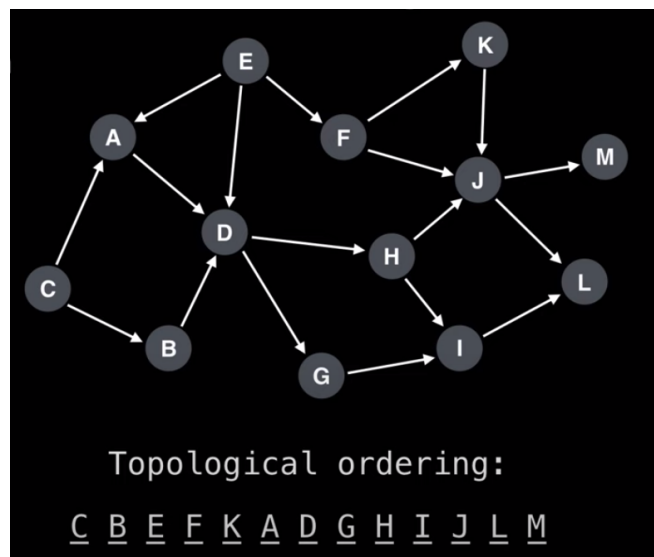
- Se comienza por la raíz y se explora cada rama lo más profundo posible antes de retroceder.
- LISTA implementada como **pila**.

Complejidades  $O(m + n)$

## Topological Sort

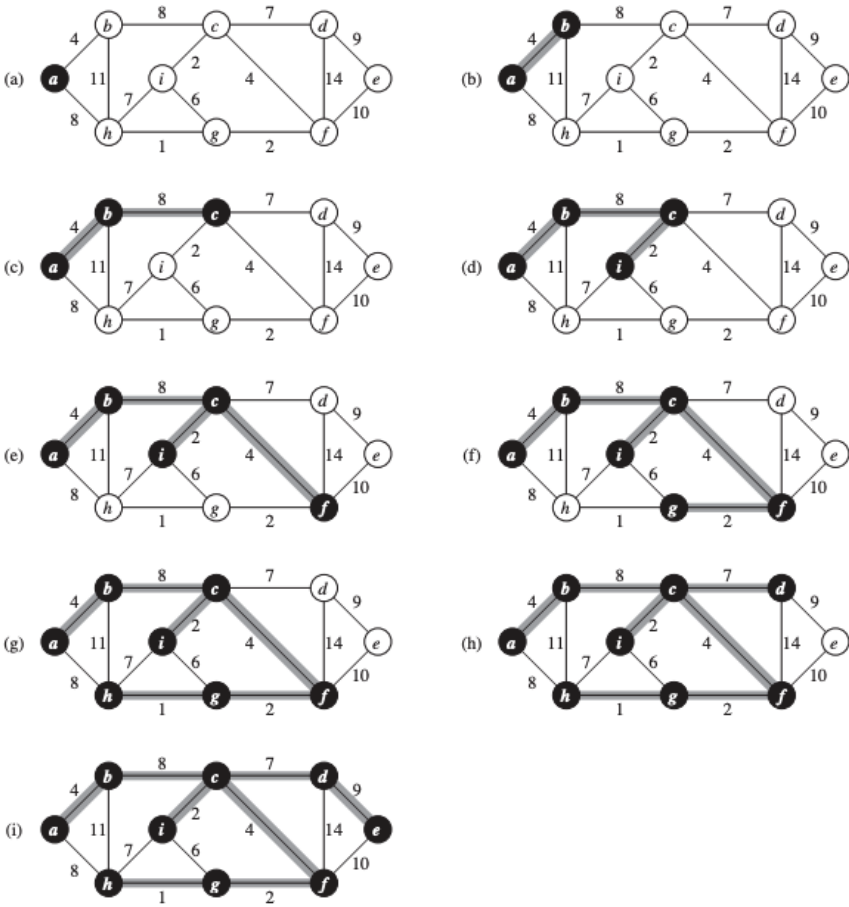
- 1 call **DFS( $G$ )** to compute finishing times  $\nu.f$  for each vertex  $\nu$
- 2 as each vertex is finished, insert it onto the front of a linked list
- 3 **return** the linked list of vertices

Complejidad:  $O(m + n)$

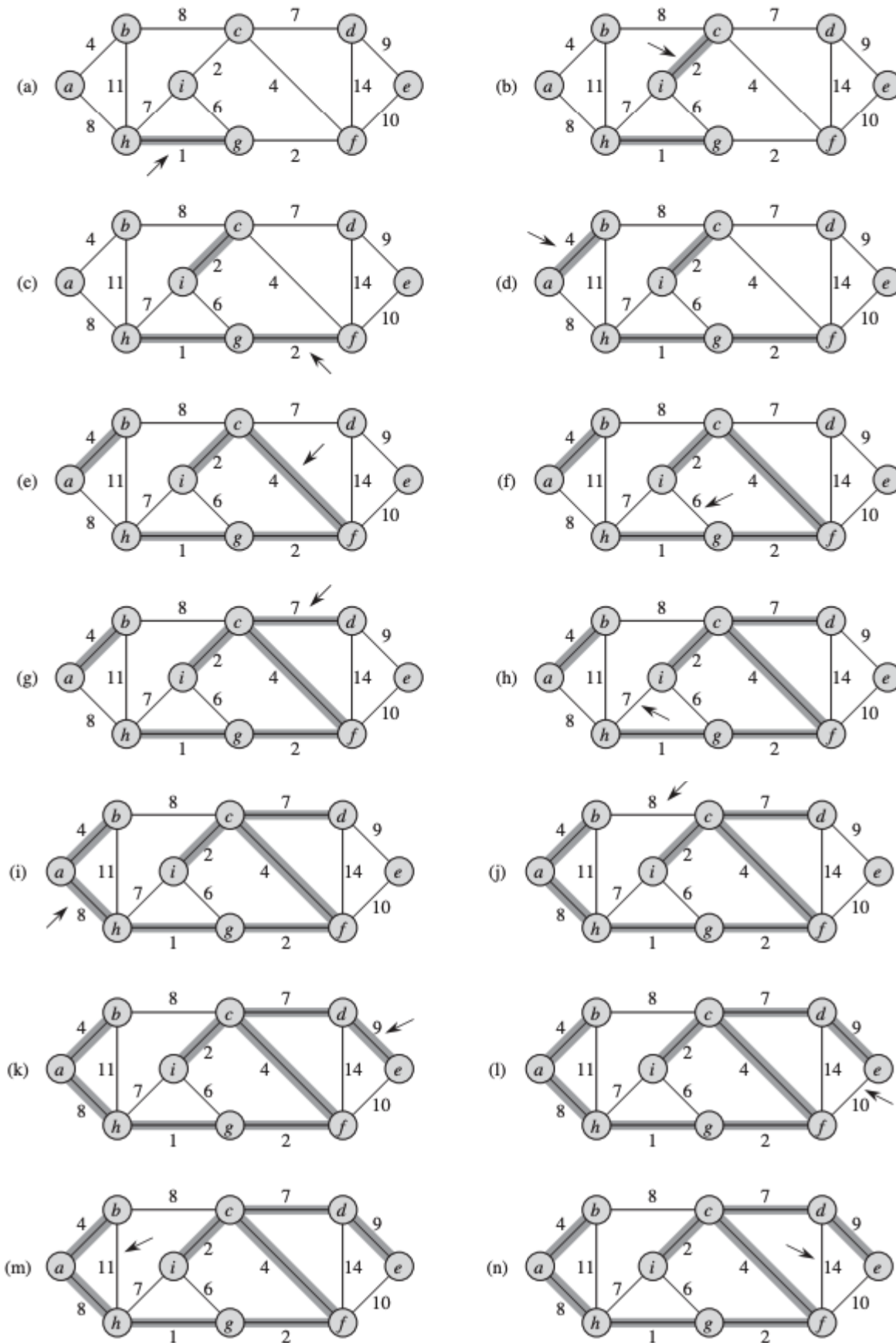


# Cálculo de (un) árbol generador mínimo

Prim



## Kruskal



## Complejidades

- $O(n^2)$  usando matriz de adyacencias  $\Rightarrow$  Preferible en grafos **densos**
- $O(m \log(n))$  usando binary heaps y lista de adyacencias  $\Rightarrow$  Preferible en grafos **rales**
- **Prim** puede mejorarse a  $O(m + n \log(n))$  usando Fibonacci Heap y lista de adyacencias.

# Single-Source Shortest Paths

En grafos no pesados  $\Rightarrow$  podemos aplicar BFS.

**Nota:** Los algoritmos estudiados acá asumen que  $G$  está guardado como lista de adyacencias y que, además, cada arista tiene guardado su peso.

Caso grafos dirigidos acíclicos (DAG)

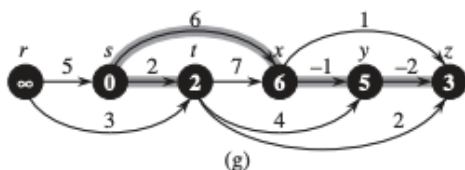
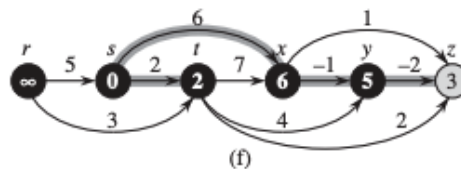
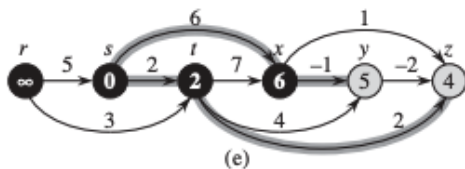
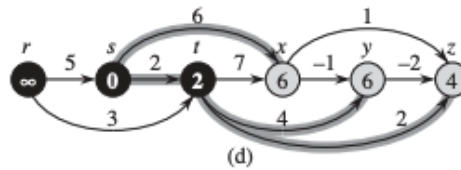
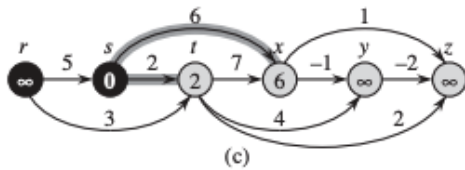
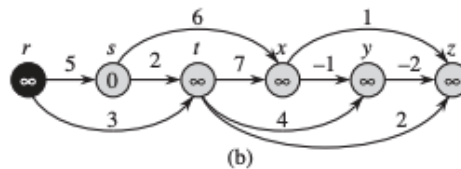
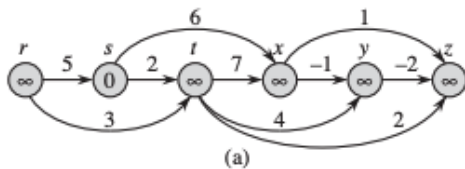
Se relajan las aristas después de hacer topological sort de sus vértices

Complejidad

-  $O(m + n)$

**DAG-SHORTEST-PATHS**( $G, w, s$ )

- 1 topologically sort the vertices of  $G$
- 2 INITIALIZE-SINGLE-SOURCE( $G, s$ )
- 3 for each vertex  $u$ , taken in topologically sorted order
- 4     for each vertex  $v \in G.Adj[u]$
- 5         RELAX( $u, v, w$ )



Critical Paths

A critical path is a longest path through the dag, corresponding to the longest time to perform any sequence of jobs. Thus, the weight of a critical path provides a lower bound on the total time to perform all the jobs.

We can find a critical path by negating the edge weights and running **DAG-SHORTEST-PATHS**.

Bellman-Ford

- Sirve en grafos con ciclos negativos.

Complejidad

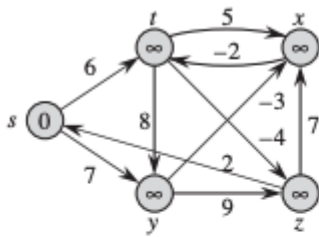
-  $O(n m)$

BELLMAN-FORD( $G, w, s$ )

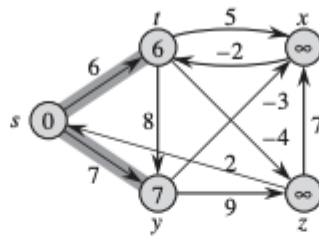
```

1 INITIALIZE-SINGLE-SOURCE( $G, s$ )
2 for  $i = 1$  to  $|G.V| - 1$ 
3   for each edge  $(u, v) \in G.E$ 
4     RELAX( $u, v, w$ )
5 for each edge  $(u, v) \in G.E$ 
6   if  $v.d > u.d + w(u, v)$ 
7     return FALSE
8 return TRUE

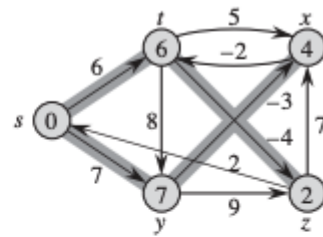
```



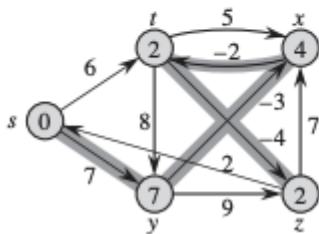
(a)



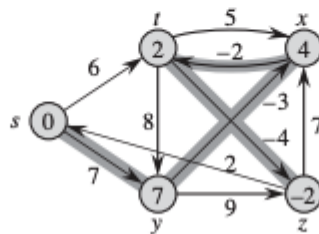
(b)



(c)



(d)



(e)

Dijkstra

- Solo aristas **no negativas**.
- Mejor tiempo que Bellman-Ford.

Complejidad

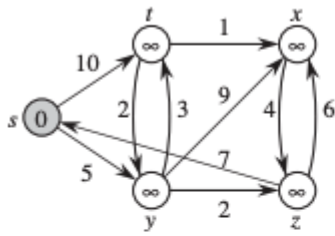
- $O(n^2)$  guardando las *distancias* de los vértices en un array.
- $O(m \log(n))$  con binary heap **si el grafo es raro** y todos los vértices son alcanzables desde el source  $s$ .
- $O(m + n \log(n))$  con Fibonacci Heap

DIJKSTRA( $G, w, s$ )

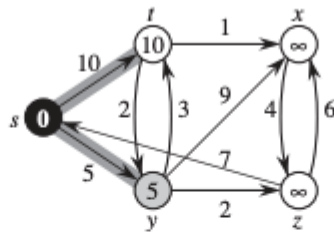
```

1 INITIALIZE-SINGLE-SOURCE( $G, s$ )
2  $S = \emptyset$ 
3  $Q = G.V$ 
4 while  $Q \neq \emptyset$ 
5    $u = \text{EXTRACT-MIN}(Q)$ 
6    $S = S \cup \{u\}$ 
7   for each vertex  $v \in G.Adj[u]$ 
8     RELAX( $u, v, w$ )

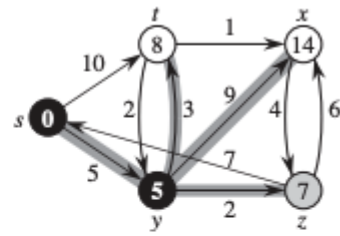
```



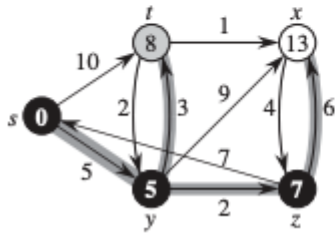
(a)



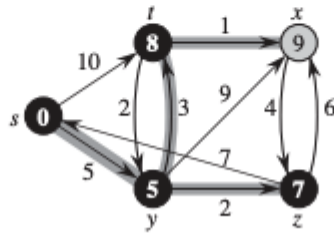
(b)



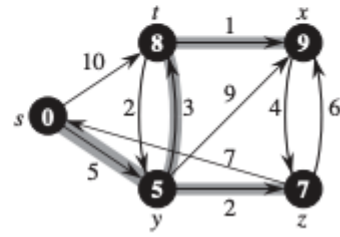
(c)



(d)



(e)



(f)

## Relaxation

**RELAX**( $u, v, w$ )

- 1 **if**  $v.d > u.d + w(u, v)$
- 2      $v.d = u.d + w(u, v)$
- 3      $v.\pi = u$

## Variantes

- **Single-destination shortest-paths**  $\Rightarrow$  Find a shortest path to a given destination vertex  $t$  from each vertex. By reversing the direction of each edge in the graph, we can reduce this problem to a single-source problem.
- **Single-pair shortest-path**  $\Rightarrow$  Find a shortest path from  $u$  to  $v$  for given vertices  $u$  and  $v$ . If we solve the single-source problem with source vertex  $u$ , we solve this problem also. Moreover, all known algorithms for this problem have the same worst-case asymptotic running time as the best single-source algorithms.
- **All-pairs shortest-paths**  $\Rightarrow$  Find a shortest path from  $u$  to  $v$  for every pair of vertices  $u$  and  $v$ . Although we can solve this problem by running a single-source algorithm once from each vertex, we usually can solve it faster.

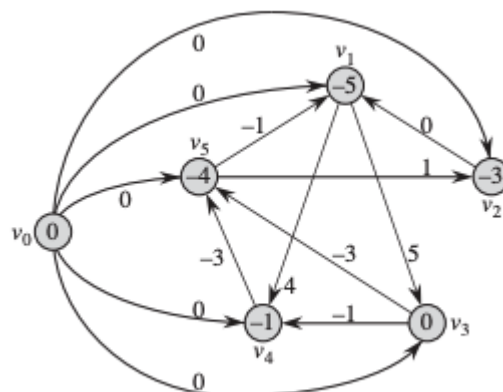
## Sistemas de diferencias

Aplicamos Bellman-Ford desde el vértice  $v_0$ .

Si el algoritmo retorna *true*, los  $\delta(v_0, v_i)$  que encontró representan una solución al sistema.

## Ejemplo

$$\begin{aligned}
 x_1 - x_2 &\leq 0, \\
 x_1 - x_5 &\leq -1, \\
 x_2 - x_5 &\leq 1, \\
 x_3 - x_1 &\leq 5, \\
 x_4 - x_1 &\leq 4, \\
 x_4 - x_3 &\leq -1, \\
 x_5 - x_3 &\leq -3, \\
 x_5 - x_4 &\leq -3.
 \end{aligned}$$



## Complejidad

- $O(n^2 + nm)$  porque el sistema genera un grafo con  $n + 1$  vértices y  $n + m$  aristas.

## All-Pairs Shortest Paths

Se asume en la mayoría de los casos que el grafo viene dado en una matriz de adyacencias donde cada entrada de la misma representa el peso de una arista.

### Dijkstra sobre cada vértice

Solo sirve si el grafo no tiene aristas negativas.

#### Complejidades

- $O(n^3)$  usando un array como cola de prioridad.
- $O(mn \log(n))$  usando un binary heap  $\Rightarrow$  es una mejora si el grafo es raro.
- $O(n^2 \log n + nm)$  usando un Fibonacci Heap.

**Observación:** En caso de tener aristas negativas, podemos aplicar el [Algoritmo de Johnson](#) que convierte primero las aristas en positivas antes de aplicar Dijkstra  $n$  veces.

### Bellman-Ford

Sirve aún con ciclos negativos.

#### Complejidades

- $O(mn^2) \Rightarrow$  se convierte en  $O(n^4)$  en grafos densos

### Floyd-Warshall

Sirve con ciclos negativos

```
Floyd(G)
entrada:  $G = (V, X)$  de  $n$  vertices
salida:  $D$  matriz de distancias de  $G$ 

 $D \leftarrow L$ 
para  $k$  desde 1 a  $n$  hacer
    para  $i$  desde 1 a  $n$  hacer
        si  $d[i][k] \neq \infty$  entonces
            si  $d[i][k] + d[k][i] < 0$  entonces
                retornar "Hay circuitos negativos"
            fin si
            para  $j$  desde 1 a  $n$  hacer
                 $d[i][j] \leftarrow \min(d[i][j], d[i][k] + d[k][j])$ 
            fin para
        fin si
    fin para
fin para
retornar  $D$ 
```

#### Complejidades

- $O(n^3)$

## Algoritmo de Johnson

Usa tanto Bellman-Ford como Dijkstra.

Se asume que el grafo está guardado como lista de adyacencias.

Reporta si se encuentra un ciclo negativo.

El algoritmo consiste en convertir todas las aristas del grafo en positivas y luego aplicar Dijkstra  $n$  veces (es decir, sirve con aristas negativas). Para convertirlo, se agrega un vértice source  $s$  y se aplica un algoritmo de SSSP desde él (por ejemplo BF). Con este peso se hace el reweight de las aristas. Luego se corre Dijkstra  $n$  veces.

### Complejidades

- $O(n^2 \log n + nm)$  usando un Fibonacci Heap.
- $O(nm \log(n))$  con Binary heap.
- Ambas son mejores complejidades que el Floyd-Warshall para grafos raros.

### JOHNSON( $G, w$ )

```
1  compute  $G'$ , where  $G'.V = G.V \cup \{s\}$ ,  
    $G'.E = G.E \cup \{(s, v) : v \in G.V\}$ , and  
    $w(s, v) = 0$  for all  $v \in G.V$   
2  if BELLMAN-FORD( $G', w, s$ ) == FALSE  
3    print "the input graph contains a negative-weight cycle"  
4  else for each vertex  $v \in G'.V$   
5    set  $h(v)$  to the value of  $\delta(s, v)$   
   computed by the Bellman-Ford algorithm  
6  for each edge  $(u, v) \in G'.E$   
7     $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$   
8  let  $D = (d_{uv})$  be a new  $n \times n$  matrix  
9  for each vertex  $u \in G.V$   
10   run DIJKSTRA( $G, \hat{w}, u$ ) to compute  $\hat{\delta}(u, v)$  for all  $v \in G.V$   
11   for each vertex  $v \in G.V$   
12      $d_{uv} = \hat{\delta}(u, v) + h(v) - h(u)$   
13  return  $D$ 
```



# Flujo máximo - Corte mínimo

Dada una red  $N = (V, X)$  con función de capacidad  $c$ , fuente  $s$  y sumidero  $t$ :

- Un **flujo factible** es una función  $f : X \rightarrow \mathbb{R}^+$  que verifica:
  - $0 \leq f(e) \leq c(e)$  para toda arista  $e \in X$ .
  - Ley de conservación de flujo:

$$\sum_{e \in In(v)} f(e) = \sum_{e \in Out(v)} f(e)$$

para todo vértice  $v \in V \setminus \{s, t\}$ , donde

$$In(v) = \{e \in X, e = (w \rightarrow v), w \in V\}$$

$$Out(v) = \{e \in X, e = (v \rightarrow w), w \in V\}$$

- El **valor del flujo** es

$$F = \sum_{e \in In(t)} f(e) - \sum_{e \in Out(s)} f(e).$$

Capacidad de un corte

Suma de las capacidades de las aristas que **salen** del corte.

Camino de aumento

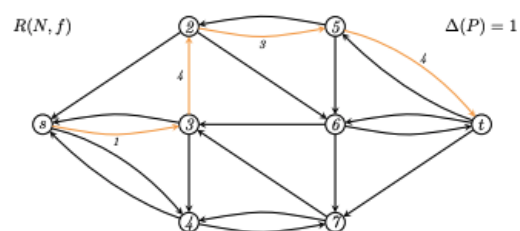
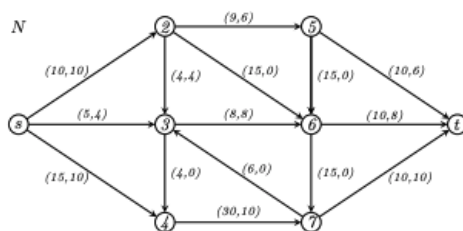
**Definición 8.** Dada una red  $N = (V, X)$  con función de capacidad  $c$ , un flujo factible  $f$  y un camino de aumento  $P$  en  $R(N, f)$ :

- Para cada arco  $(v \rightarrow w)$  de  $P$ , definimos

$$\Delta((v \rightarrow w)) = \begin{cases} c((v \rightarrow w)) - f((v \rightarrow w)) & \text{si } (v \rightarrow w) \in X \\ f((w \rightarrow v)) & \text{si } (w \rightarrow v) \in X \end{cases}$$

- $\Delta(P) = \min_{e \in P} \{\Delta(e)\}$

**Ejemplo 6.**



```
Ford&Fulkerson( $N$ )
  entrada:  $N = (V, X)$  con función de capacidad  $c$ 
  salida:  $f$  flujo máximo
  definir un flujo inicial en  $N$ 
  (por ejemplo  $f(e) \leftarrow 0$  para todo  $e \in X$ )
  mientras exista  $P$  camino de aumento en  $R(N, f)$  hacer
    para cada arco  $(v \rightarrow w)$  de  $P$  hacer
      si  $(v \rightarrow w) \in X$  entonces
         $f((v \rightarrow w)) \leftarrow f((v \rightarrow w)) + \Delta(P)$ 
      si no comentario:  $((w \rightarrow v) \in X)$ 
         $f((w \rightarrow v)) \leftarrow f((w \rightarrow v)) - \Delta(P)$ 
      fin si
    fin para
  fin mientras
```

- A lo sumo  $n * U$  iteraciones, que es el valor del flujo máximo.

Complejidad

- $O(mF) = O(m * nU) \Rightarrow$  donde  $F$  es el valor del flujo máximo.  $U$  es una cota superior finita para el valor de las capacidades (si los valores del flujo inicial y las capacidades de los arcos son enteros).

### Algoritmo de Edmonds y Karp

- Buscar el camino de aumento en la red residual con BFS.
- A lo sumo  $n * m$  iteraciones.

Complejidad

- $O(m * nm) = O(nm^2)$  (siempre que sean capacidades enteras)

## Flujo máximo de costo mínimo

Cancelación de ciclos

### Algoritmo de cancelación de ciclos

1. Establecer un flujo  $x$  factible.
2. Mientras  $G_x$  contenga un ciclo negativo  $W$  hacer
  - ▶ Definir  $\delta := \min\{r_{ij} : ij \in W\}$ .
  - ▶ Aumentar  $\delta$  unidades de flujo a lo largo del ciclo  $W$  y actualizar  $x$ .
3. Fin mientras

- El flujo factible se puede hallar con **E&K**.
- Los ciclos negativos se encuentran con Bellman-Ford  $\Rightarrow$  complejidad  $O(mn)$
- Cantidad máxima de iteraciones:  $mCU$

► ¿Cuál es la complejidad computacional de este algoritmo?

1.  $C := \max\{c_{ij} : ij \in A\}.$
2.  $U := \max\{u_{ij} : ij \in A\}.$

- El costo del flujo inicial no puede ser superior a  $mCU$  y el costo final no puede ser inferior a cero. Luego, el algoritmo realiza a lo sumo  $mCU$  iteraciones y su complejidad total es  $O(nm^2CU)$ .

## Successive Shortest Path

Consiste en usar **E&K** pero al elegir el camino de aumento en la red residual se toma aquel de costo mínimo.

### Successive Shortest Path

```

1 Transform network G by adding source and sink
2 Initial flow x is zero
3 while ( Gx contains a path from s to t ) do
4   Find any shortest path P from s to t
5   Augment current flow x along P
6   update Gx

```

Costo mínimo

- El camino de costo mínimo se puede hallar con Dijkstra por ejemplo modificando la función que usa el algoritmo para determinar el "shortest" path.

Costo:

Costo por iteración:  $O(nm)$  Bellman Ford  
(se puede modificar para usar Dijkstra)

cantidad de iteraciones:  $O(mU)$

de  $s$  salen a lo sumo  $m$  aristas  
cada una con capacidad a lo sumo  $U$ .  
cada iteración aumenta el flujo

$\Rightarrow O(nm^2U)$

Para asignación  $U=1 \Rightarrow$  costo  $O(nm^2)$

se puede bajar a  
 $O(nm + m^2 \lg m) \in O(n^3)$

Aluja Pg. 500

- $O(m n^2 U)$  usando Bellman-Ford

## Complejidad

### Problemas NP-completos

Un problema de decisión  $\Pi$  es **NP-completo** si:

1.  $\Pi \in \text{NP}$
2.  $\forall \bar{\Pi} \in \text{NP}, \bar{\Pi} \leq_p \Pi$

Si un problema  $\Pi$  verifica la condición 2.,  $\Pi$  es **NP-difícil** (es al menos tan **difícil** como todos los problemas de NP).

Es decir, para demostrar que un problema  $\Pi_1$  es NP-completo hay que demostrar las siguientes 2 cosas:

- El problema pertenece a NP  $\Rightarrow$  una respuesta es verificable en tiempo polinomial con un certificado y un verificador.
- Reducimos otro problema  $\Pi_2$  que sabemos que es NP-completo a él, demostrando que  $\Pi_1$  es tan o más difícil que

$$\Pi_2 \Rightarrow \Pi_2 \leq_p \Pi_1$$

► **Proposición.** Si  $\Pi \in \text{P}$ , entonces  $\Pi^c \in \text{P}$ .

► **Proposición,** Si  $\Pi \in \text{NP}$ , entonces  $\Pi^c \in \text{Co-NP}$ .

### co-NP

- $\Pi \in \text{coNP} \Leftrightarrow \bar{\Pi} \in \text{NP}$
- Si  $P$  es NP-completo, por definición  $\bar{P}$  está en co-NP-completo.

Si  $P = NP$  entonces  $NP = coNP$ .

- Verdadero
- Si  $P = NP$  entonces  $P = NP \subseteq coNP$ . Falta ver que  $coNP \subseteq NP$ .
- Si  $\Pi \in coNP$  entonces  $\bar{\Pi} \in NP$ , y por hipótesis  $\bar{\Pi} \in P$ . Entonces puedo decidir  $\bar{\Pi}$  en tiempo polinomial, y por lo tanto también puedo decidir  $\Pi$  en  $P$ . Finalmente, como  $P \subseteq coNP$ , tengo que  $\Pi \in coNP$ .

### Observación

Si  $P$  es NP-completo, por definición  $\bar{P}$  está en co-NP-completo.

### Demostración.

Sea  $A \in co-NP \Rightarrow \bar{A} \in NP$ .

Entonces, como  $P$  es NP-completo, existe una reducción polinomial  $f$  tal que  $x \in \bar{A} \Leftrightarrow f(x) \in P$ .

Esto es equivalente a decir que  $x \notin \bar{A} \Leftrightarrow f(x) \notin P$ , y esto a su vez equivale a

$$x \in A \Leftrightarrow f(x) \in \bar{P}$$

$\Rightarrow f$  también es una reducción polinomial de  $A$  a  $\bar{P}$ , o dicho de otro modo,  $A \leq_p \bar{P}$ . Por lo tanto,  $\bar{P}$  es co-NP-completo.

